AI RESEARCH SERIES

AIBench: Cost–Quality Frontiers in LLM Code Generation

A machine-verified evaluation of seven large language models on graded React and Rust engineering tasks, with an analysis of retry strategies and per-task cost economics.

Publication date
June 2026

Benchmark run
June 9, 2026

Authors
AtomicoLabs Research

Contact
c@atomicolabs.com

Abstract

We present results from AIBench, an internally developed benchmark that evaluates large language models (LLMs) on ten machine-verified code generation tasks across two technology stacks: React/TypeScript (frontend engineering) and Rust (systems programming). Seven model configurations were evaluated under a uniform protocol permitting up to three attempts per task, with full cost accounting including failed attempts. Claude Fable 5 achieved a perfect score (10/10) at a total cost of $2.38; DeepSeek V4 Flash passed 8/10 tasks at $0.015 — a 159× total-cost difference for a two-task quality gap. Our most consequential finding concerns evaluation protocol rather than model choice: enabling verification-driven retries raised Claude Sonnet 4.6 from 3/10 to 9/10 on identical tasks, tripling its pass rate at comparable cost per success. We argue that benchmark scores reported without an attached retry policy are not directly actionable, and that verification-plus-retry loops should be treated as baseline architecture in production code generation systems. We close with deployment recommendations, threats to validity, and planned extensions of the benchmark.

Contents

1 Introduction

Public coding benchmarks increasingly fail to answer the question practitioners actually face: *which model should run inside a production code generation loop, and at what cost?* Headline leaderboard scores typically obscure three variables that dominate real-world outcomes: the verification regime (was the output actually executed and tested?), the retry policy (how many attempts were allowed, and with what feedback?), and the full cost of obtaining a success, including the cost of failures along the way.

AIBench is our attempt to measure what we deploy. The benchmark uses tasks distilled from production work at AtomicoLabs — component construction, form validation, CRUD assembly, concurrent I/O, lock-free data structures — graded exclusively by automated harnesses. No human judgment participates in scoring. Every run is accounted at the API-invoice level, so a model's cost reflects what an engineering team would actually pay, not a per-token abstraction.

This report covers the June 9, 2026 run: seven model configurations, ten tasks, two language suites.

2 Benchmark Design

2.1 Task suites

Tasks are organized in two suites of five, each ordered by increasing difficulty (L1–L5). Difficulty levels are calibrated so that L1 should be solvable by any competent code model, while L5 requires sustained multi-file reasoning or specialized domain knowledge.

TABLE I — TASK SUITE COMPOSITION

Level	React suite	Rust suite	Primary capability tested
L1	Stat card component	Merge intervals	Basic synthesis from spec
L2	Login form w/ validation	Word-count CLI (wc clone)	State handling, I/O, edge cases
L3	Sortable users table	LRU cache	Data structures, interaction logic
L4	Posts CRUD interface	Concurrent HTTP fetcher	Multi-component / async coordination
L5	Multi-page application	SPSC lock-free ring buffer	Architecture / memory ordering

The React suite emphasizes component composition, controlled state, form handling, and routing. The Rust suite escalates from algorithmic warm-ups into ownership-heavy territory: the L4 fetcher requires correct async task coordination and error propagation, while the L5 single-producer single-consumer ring buffer demands correct use of atomics and acquire/release memory ordering — a known failure mode for code models, which tend to produce superficially plausible but subtly racy implementations.

2.2 Verification harness

Each task ships with an automated grader. A submission passes only if all of the following hold:

- **Compilation / type-check:** the artifact builds cleanly (tsc for React tasks, cargo build for Rust tasks).

- **Unit and behavioral tests:** a hidden test set exercises functional requirements, including edge cases not spelled out in the prompt.
- **Task-specific checks:** e.g., the ring buffer is exercised under concurrent load; the CRUD interface is driven through create/edit/delete flows.

Binary pass/fail per task; no partial credit. This is deliberately conservative: in production, code that almost works still requires human time.

3 Evaluation Protocol

3.1 Trials and retries

Each model received one trial per task with up to three attempts. On failure, the model received the harness output (compiler errors, failing test names and assertions) and could revise its solution. Two configurations of Claude Sonnet 4.6 were run — single-pass and with retries — to isolate the effect of the retry loop itself; Claude Fable 5's single-attempt subscore was additionally recorded for the same purpose.

3.2 Cost accounting

Costs are total API spend per suite, including all failed attempts, measured at provider list prices on the run date. We report two derived metrics: **total cost** (the invoice) and **cost per passed task** (total cost divided by passes), the latter being the figure most relevant to production budgeting. Per-test cost breakdowns were captured for the four primary configurations; three budget-tier runs recorded suite-level totals only.

3.3 Models under test

TABLE II — EVALUATED CONFIGURATIONS

Configuration	Tier	Retry policy
Claude Fable 5	Frontier	Up to 3 attempts
Claude Opus 4.5	Frontier	Up to 3 attempts
Claude Sonnet 4.6 (retries)	Mid	Up to 3 attempts
Claude Sonnet 4.6 (1-pass)	Mid	Single attempt
DeepSeek V4 Pro	Budget	Single attempt
DeepSeek V4 Flash	Budget	Up to 3 attempts
Tencent HY3 Preview	Budget	Single attempt

4 Results

4.1 Aggregate pass rates and cost

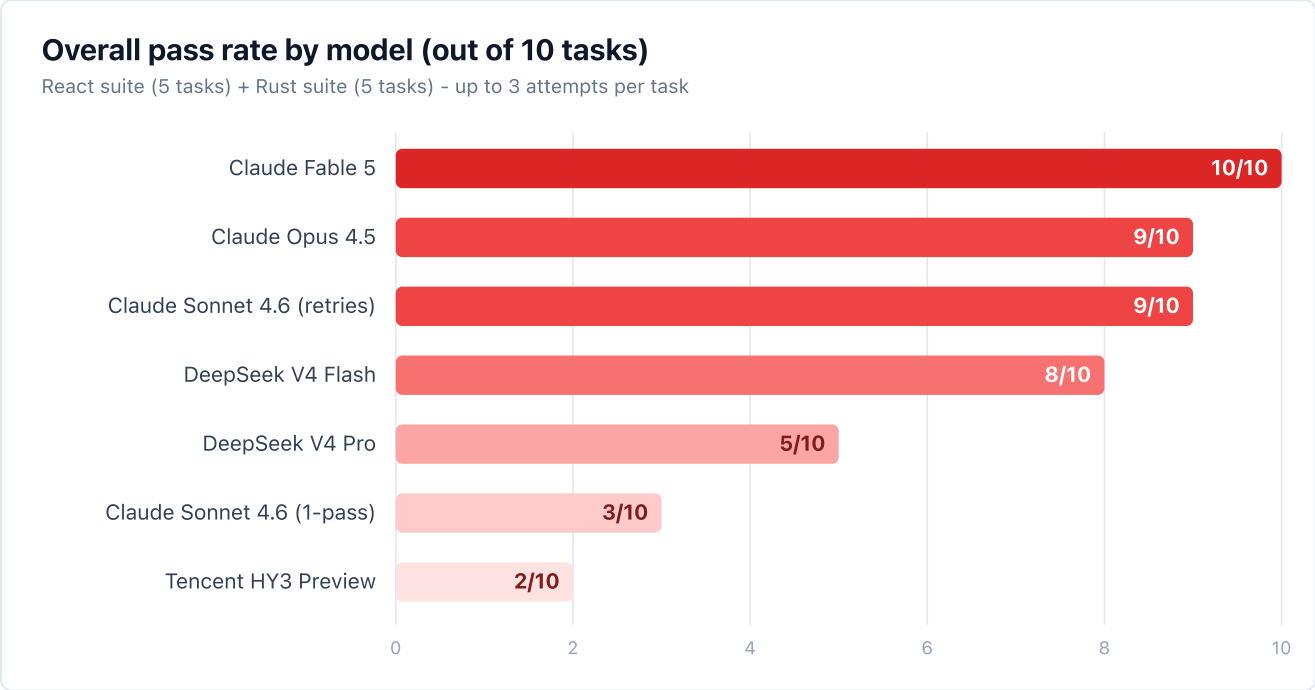

FIGURE 1 — Overall pass rate by model across both suites (10 tasks, up to 3 attempts each).

TABLE III — SUMMARY OF RESULTS (JUNE 9, 2026 RUN)

Model	React	Rust	Total	React cost	Rust cost	Total cost
Claude Fable 5	5/5	5/5	**10/10**	$0.65	$1.73	$2.38
Claude Opus 4.5	4/5	5/5	**9/10**	$0.42	$0.58	$1.00
Claude Sonnet 4.6 (retries)	4/5	5/5	**9/10**	$0.28	$0.47	$0.75
DeepSeek V4 Flash	4/5	4/5	**8/10**	$0.005	$0.010	$0.015
DeepSeek V4 Pro	3/5	2/5	**5/10**	$0.02	$0.08	$0.11
Claude Sonnet 4.6 (1-pass)	1/5	2/5	**3/10**	$0.09	$0.18	$0.27
Tencent HY3 Preview	2/5	0/5	**2/10**	$0.003	$0.016	$0.02

4.2 Per-task results, frontier configurations

TABLE IV — CLAUDE FABLE 5, PER-TASK DETAIL (10/10, $2.38)

Suite	Task	Result	Attempts	Cost
React	L1-statcard	Pass	1	$0.03
React	L2-loginform	Pass	1	$0.05
React	L3-userstable	Pass	2	$0.18
React	L4-postscrud	Pass	1	$0.18
React	L5-pagestoapp	Pass	3	$0.20
Rust	L1-merge-intervals	Pass	1	$0.06
Rust	L2-wc-rs	Pass	1	$0.11
Rust	L3-cache	Pass	1	$0.19
Rust	L4-fetcher	Pass	2	$1.06
Rust	L5-spsc-ring	Pass	1	$0.30

TABLE V — CLAUDE OPUS 4.5, PER-TASK DETAIL (9/10, $1.00)

Suite	Task	Result	Attempts	Cost
React	L1-statcard	Pass	1	$0.01
React	L2-loginform	Pass	1	$0.02
React	L3-userstable	Pass	2	$0.07
React	L4-postscrud	Fail	3	$0.26
React	L5-pagestoapp	Pass	3	$0.06
Rust	L1-merge-intervals	Pass	1	$0.03
Rust	L2-wc-rs	Pass	2	$0.09
Rust	L3-cache	Pass	1	$0.07
Rust	L4-fetcher	Pass	2	$0.20
Rust	L5-spsc-ring	Pass	2	$0.19

TABLE VI — CLAUDE SONNET 4.6 WITH RETRIES, PER-TASK DETAIL (9/10, $0.75)

Suite	Task	Result	Attempts	Cost
React	L1-statcard	Pass	1	$0.01
React	L2-loginform	Pass	2	$0.04
React	L3-userstable	Pass	2	$0.04
React	L4-postscrud	Fail	3	$0.17
React	L5-pagestoapp	Pass	2	$0.02
Rust	L1-merge-intervals	Pass	1	$0.02
Rust	L2-wc-rs	Pass	2	$0.05
Rust	L3-cache	Pass	1	$0.09
Rust	L4-fetcher	Pass	2	$0.17
Rust	L5-spsc-ring	Pass	3	$0.14

TABLE VII — DEEPSEEK V4 FLASH, PER-TASK DETAIL (8/10, $0.015)

Suite	Task	Result	Attempts	Cost
React	L1-statcard	Pass	1	$0.0002
React	L2-loginform	Pass	1	$0.0003
React	L3-userstable	Pass	3	$0.0011
React	L4-postscrud	Fail	3	$0.0031
React	L5-pagestoapp	Pass	1	$0.0004
Rust	L1-merge-intervals	Pass	1	$0.0002
Rust	L2-wc-rs	Pass	2	$0.0014
Rust	L3-cache	Pass	3	$0.0021
Rust	L4-fetcher	Pass	3	$0.0024
Rust	L5-spsc-ring	Fail	3	$0.0039

4.3 Cost efficiency

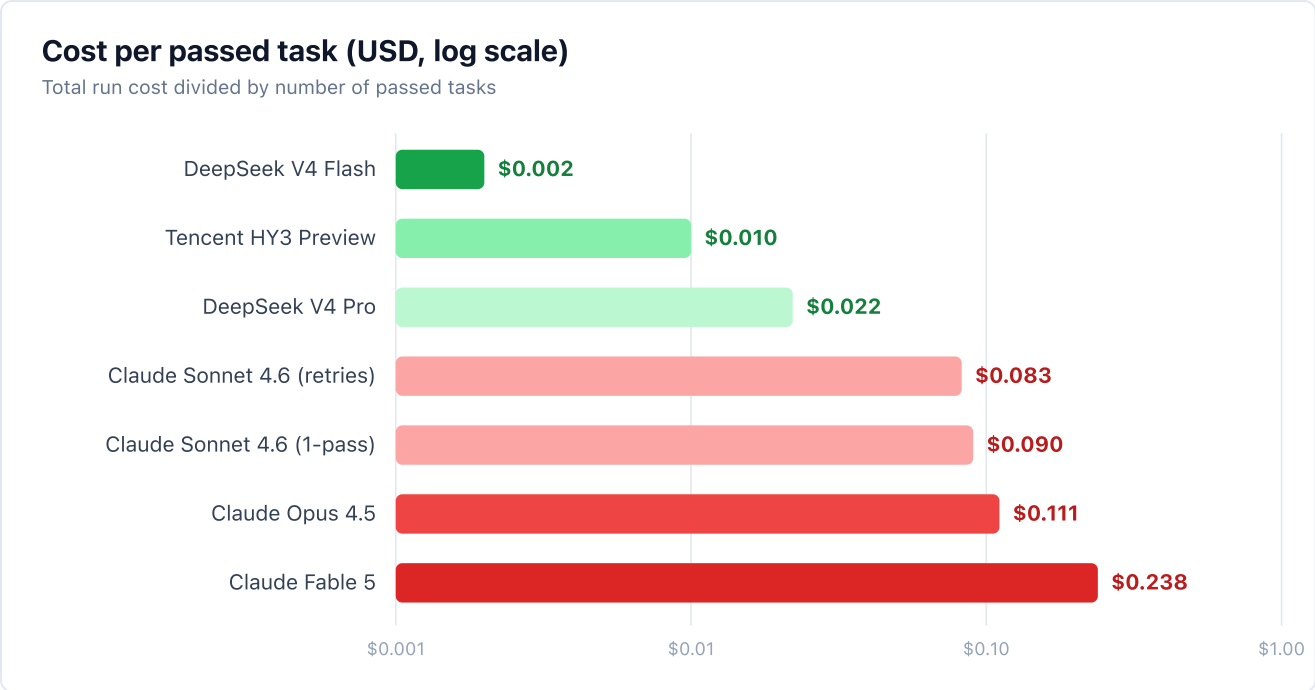

FIGURE 2 — Cost per passed task in USD (log scale). Failed attempts are included in numerator.

TABLE VIII — COST PER PASSED TASK

Model	Passes	Total cost	Cost / pass
DeepSeek V4 Flash	8	$0.015	**$0.002**
Tencent HY3 Preview	2	$0.02	$0.010
DeepSeek V4 Pro	5	$0.11	$0.022
Claude Sonnet 4.6 (retries)	9	$0.75	$0.083
Claude Sonnet 4.6 (1-pass)	3	$0.27	$0.090
Claude Opus 4.5	9	$1.00	$0.111
Claude Fable 5	10	$2.38	$0.238

4.4 The retry effect

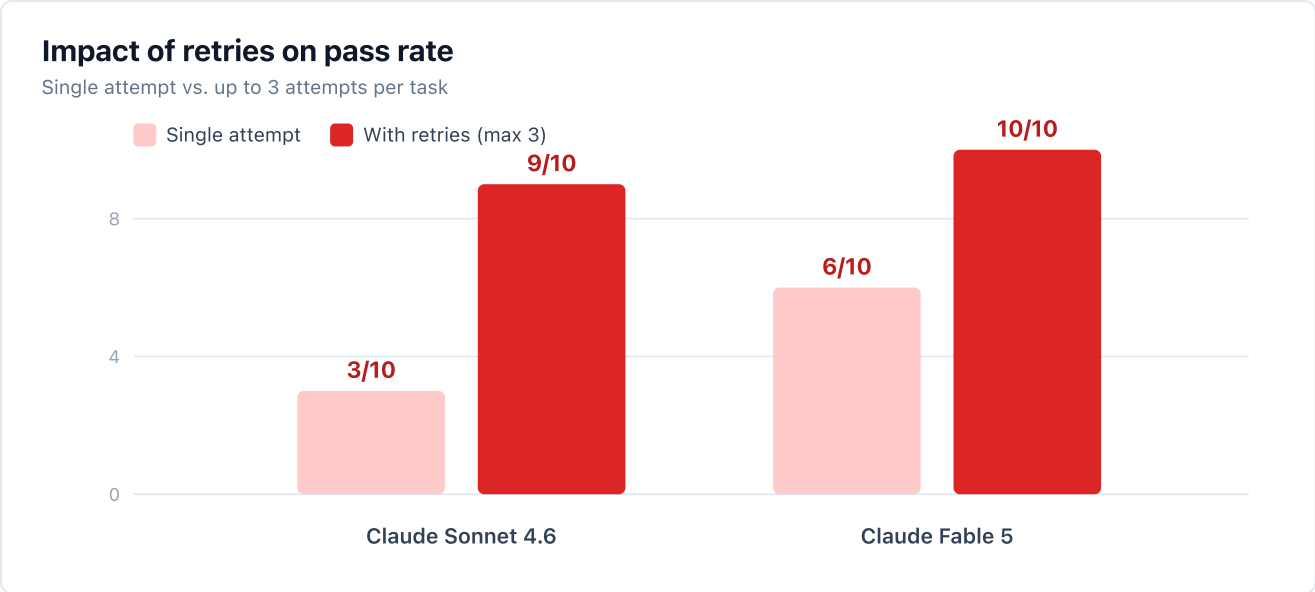

FIGURE 3 — Pass rates with and without verification-driven retries (max 3 attempts).

TABLE IX — RETRY IMPACT

Model	Single attempt	With retries	Improvement
Claude Sonnet 4.6	3/10	9/10	**+6 passes**
Claude Fable 5	6/10	10/10	**+4 passes**

5 Analysis

5.1 The frontier is tight, and one task defines it

The top three configurations are separated by a single task: the React L4 CRUD interface, which only Claude Fable 5 completed. All three frontier configurations cleared the full Rust suite, including the L5 lock-free ring buffer — historically the most failure-prone task in our harness. The L4 CRUD failure mode was consistent across Opus 4.5 and Sonnet 4.6: both models produced structurally correct interfaces that failed behavioral checks on edit-flow state synchronization, and neither recovered within the three-attempt budget. This suggests the gap is a genuine capability difference in multi-component state reasoning rather than sampling noise, though multi-trial confirmation is pending (Section 7).

5.2 Budget models are one task behind, at 1/100th the cost

DeepSeek V4 Flash passed 8/10 for $0.015 total — trailing the $1.00–$2.38 frontier runs by one to two tasks. Its two failures (React L4, Rust L5) were precisely the two hardest tasks in the benchmark. For pipeline architectures that route first attempts to a cheap model and escalate failures to a frontier model, these results imply an expected cost reduction approaching two orders of magnitude on the ~80% of tasks the budget model handles, with no quality loss on the remainder.

The same is not true across the budget tier generally: DeepSeek V4 Pro (5/10) and Tencent HY3 Preview (2/10) were run single-pass, and HY3 passed zero Rust tasks. Tier labels and price are weak predictors of capability; suite-specific evaluation remains necessary.

5.3 Retries are worth more than a model upgrade

The controlled Sonnet 4.6 comparison isolates the retry loop as the single highest-leverage variable in the study. Three attempts with harness feedback took Sonnet from 3/10 to 9/10 — a larger improvement than any model substitution at any price point in this run. Moreover, single-pass operation was *more expensive per success* ($0.090 vs $0.083): failed runs are pure cost.

> **Key finding.** A benchmark score without an attached retry policy is not actionable. Verification-plus-retry is not an optimization; it is baseline architecture for production code generation, and it changes both the quality ranking and the cost ranking of models.

Both retry-enabled Claude configurations converged near their ceiling within three attempts, indicating that most single-pass failures are recoverable errors — missed edge cases, misread specifications — rather than hard capability limits. The failures that persisted through all three attempts (React L4 for Opus/Sonnet, React L4 and Rust L5 for Flash) are correspondingly more informative about true capability boundaries than any single-attempt statistic.

5.4 Rust separates the field

Aggregate React scores compress the field (5/5 to 1/5), but Rust scores split it cleanly: frontier configurations went 15/15 collectively, while budget-tier models dropped 9 of 15. Ownership, lifetimes, and concurrency primitives appear to function as a capability watermark that cheaper models have not yet crossed — consistent with our qualitative observation that budget-model Rust failures were dominated by borrow-checker errors and incorrect atomic orderings rather than logic mistakes.

6 Deployment Recommendations

1. **Cost-sensitive workloads:** DeepSeek V4 Flash as first line (8/10 at $0.015, ≈159× cheaper than the top scorer), with frontier escalation on failure.
2. **Maximum quality:** Claude Fable 5 — the only configuration to clear all ten tasks, including the L4 CRUD task that defeated every other model.
3. **Best balance:** Claude Sonnet 4.6 with retries: matches Opus 4.5's 9/10 at 25% lower cost.
4. **Always enable verification-driven retries.** The measured effect (+4 to +6 passes out of 10) exceeds the effect of any model tier upgrade in this study, at a fraction of the price.
5. **Evaluate per stack.** React performance does not predict Rust performance below the frontier tier. Benchmark on the stack you ship.

7 Threats to Validity

- **Single trial per task.** Pass/fail outcomes at temperature carry run-to-run variance that a single trial cannot quantify. Rankings separated by one task should be read as provisional.
- **Incomplete cost telemetry.** Per-test costs were not captured for DeepSeek V4 Pro, Tencent HY3 Preview, or the Sonnet single-pass run; suite-level totals are accurate but attempt-level analysis is unavailable for those configurations.
- **Heterogeneous retry policies.** DeepSeek V4 Pro and Tencent HY3 were evaluated single-pass; given Section 5.3, their scores likely understate retry-enabled performance.
- **Task provenance.** Tasks derive from our production work and may not represent other engineering domains (data engineering, mobile, embedded).
- **Price volatility.** Cost figures reflect provider list prices on June 9, 2026 and shift frequently; ratios are more durable than absolute figures.

8 Future Work

Planned extensions: (i) multi-trial runs with bootstrap confidence intervals on pass rates; (ii) additional suites in Python and Go; (iii) wall-clock latency measurement alongside cost, enabling throughput-aware routing decisions; (iv) uniform retry policies across all tiers; and (v) a public release of task specifications and grading criteria for external reproduction.

9 About AtomicoLabs

AtomicoLabs is an AI lab that experiments, builds, and ships AI products — for its own portfolio (Super, AWEsome, Kani, Reactor) and for companies ready to turn AI into deployed software. AIBench is developed internally to ground our model selection and agent architecture decisions in machine-verified evidence.

Methodology questions and requests for specific model evaluations: **c@atomicolabs.com** · atomicolabs.com

---